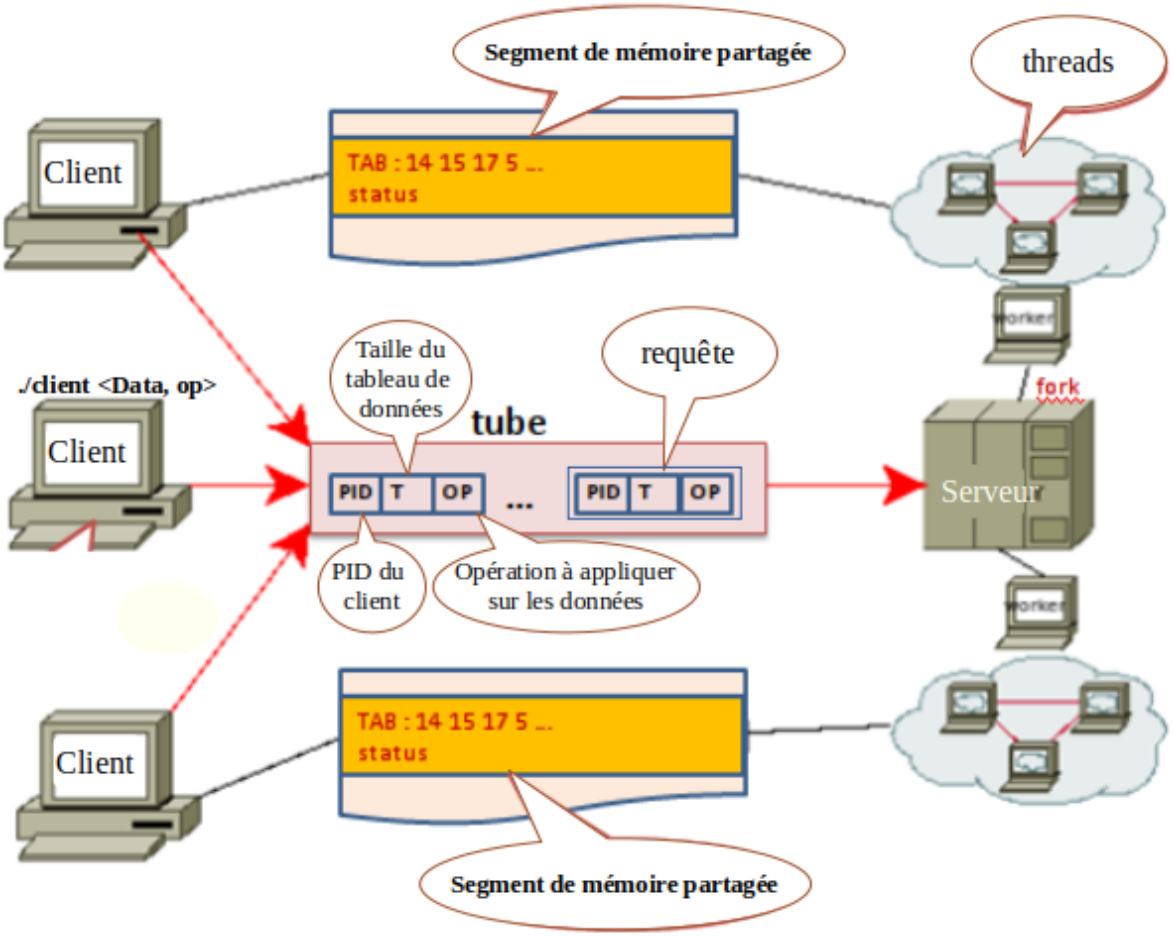
Application client/serveur de calculs de sommes préfixes - Manuel technique

Travail réalisé par :
- BOUZIDI Louisa
- DIA Modou Ndiar

Présentation de l’application

Cette application met en œuvre une architecture **Client/Serveur** permettant à un client d'envoyer des requêtes de calcul au serveur. Ce dernier se charge de traiter ces requêtes et de rendre le résultat au client qui, à son tour, procède à l’affichage du résultat. Bien évidemment, une seule instance du serveur sera en exécution à un moment donné, mais il est possible d’avoir plusieurs clients qui sollicitent des calculs de ce serveur. Une requête consiste en la demande de calcul des sommes préfixes d'un tableau d'entiers au serveur via un segment de mémoire partagée. Ce calcul peut être toute opération associative comme (+, x, min, max, pgcd, ...). Les données sont donc constituées d’un tableau de valeurs sauvegardées dans un fichier. La requête est composée du PID du client, de la taille du tableau de données et de l'opération qui devrait être appliquée sur les données. La requête est transmise au serveur via un tube (pipe) nommé qui est créé par le serveur. Les clients écrivent leurs requêtes dans ce tube alors le serveur lit chacune de ces requêtes et les traitent.



Fichiers du projet

L'application est composée des fichiers suivants:

- 3 fichiers sources représentant 3 programmes:
  - un programme implémentant les traitements du côté du client
  - un programme implémentant les traitements du côté du serveur
  - un programme permettant de contrôler le serveur : lancement, arrêt tout en prenant les précautions nécessaires pour éviter de lancer plusieurs processus serveurs en même temps et nettoyage au moment de l'arrêt du serveur et du contrôleur.
- un fichier de configuration "conf.h" partagée par les 3 fichiers précédents
- un tube créé lors du démarrage du serveur
- un makefile facilitant le processus d'installation de l'application
- 2 fichiers de documentation destinées à l'utilisateur et au développeurs.

Le programme côté client (client.c)

Ce programme permet, en résumé, de construire une requête, de récupérer des données et de les partager avec le serveur et d'envoyer, à ce dernier, cette requête et enfin d'afficher le résultat de son traitement une fois réalisé par le serveur. Le tableau ci-dessous donne une description plus détaillée de ces traitements:

Phase	Etapes
Vérification et de récupération des données	Étape 1 : Vérification et récupération des arguments fournis en ligne de commande. Dans cette étape, le client s'assure que l'utilisateur a fourni (en ligne de commande) 2 arguments : le nom (éventuellement le chemin) du fichier de données et le numéro de l'opération à appliquer sur ces données. Dans le cas d'erreurs de saisie, le programme explique à l'utilisateur le mode d'usage de l'application et s'arrête.
	Étape 2 : Récupération des données depuis le fichier dont le nom est fourni en ligne de commande. Ce fichier comporte un tableau de valeurs. Dans cette étape, on calcule aussi la taille de ce tableau de données. On vérifie que le fichier de données existe et on l'ouvre en mode lecture. Pour récupérer les données, on suppose que le fichier est composé d'une séquence de valeurs séparées par des espaces. Une vérification de ces données est effectuée et en cas d'erreur, le programme est arrêté.
Partage des données (partage du tableau de données et d'un champ de synchronisation "status")	Étape 3 : Création d'un segment de mémoire partagée qui va servir pour communiquer ce tableau de données aux processus "worker" se trouvant au niveau du serveur et qui sera associé au client. A noter que cette mémoire partagée va contenir une structure composée de 2 champs : le tableau de données et un champ d'état nommé "status" qui va être utilisé pour synchroniser le client et le processus "worker" du côté du serveur.
	Étape 4 : Transfert des données vers le segment de mémoire partagée. Une fois que ce transfert est terminé, le client va indiquer ce fait dans le champ "status" du segment de mémoire partagé en le mettant à la constante "FIN_DEPOT_DATA".
Création d'une requête, et obtention de	Étape 5 : Création d'une requête. On met 3 champs : Le PID du client, la taille du tableau de données et l'opération qui doit être appliquée par le processus "worker" sur les données

l’accès en écriture vers un tube connu du client et du serveur et écriture de la requête dans ce tube.	Étape 6 : Tentative d'ouverture en écriture du tube de communication partagé avec le serveur afin de lui transmettre une requête. Si échec --> message d'erreur et quitter
	Etape 7 : Ecriture de la requête dans le tube
Phase finale : Attente puis affichage du résultat	Étape 8 : Boucle d'attente de la fin du calcul du côté du serveur et de la remise du résultat par les workers. Le client se rend compte que le résultat est disponible en consultant le champ “status” du segment de mémoire partagée, en particulier, dès que ce champ est modifié par le serveur en y mettant la valeur “FIN_REMISE_RESULTATS”.
	Etape 9 : Affichage du résultat (du tableau de données se trouvant dans le segment de mémoire partagée)

Le serveur

Ce programme crée un tube de communication nommé (s'il n'existe pas) et rentre dans une boucle infinie dans laquelle il:

1. lit une requête depuis le tube
2. se clone et crée (grâce à l'appel fork) un processus fils (nommé worker) qui fait appel à une fonction "***traitementWorker***" ,
3. attend la fin de l'exécution du worker (son fils) pour reboucler et traiter une nouvelle requête

```
19  int main(int argc, char *argv[]) {
20
21      creerTube();  // Création du tube
22
23      // Récupération du descripteur du tube en lecture
24      // FIFO_NAME contient le nom du tube partagé entre le client et le serveur
25      // On suppose que le client et le serveur sont dans le même répertoire
26      // ************************************************************************
27
28      int fdread = 0;
29      if ((fdread = open(FIFO_NAME, O_RDONLY)) == -1) {
30          fprintf(stderr, "Impossible d'ouvrir le tube en lecture: %s\n",
31                  strerror(errno));
32          exit(EXIT_FAILURE);
33      }
34
35      // boucle infinie de lecture des requêtes depuis le tube
36      // *****************************************************
37      struct requete req;
38      while (1) {
39          while (1) {
40              if (read(fdread, &req, sizeof(req)) > 0) break;
41          }
42          pid_t worker= fork();
43          if(worker==0){
44              traitementWorker(req.pid, req.dataSize, req.operation);
45          }
46      }
47      return 0;
48  }
```

La fonction "***traitementWorker***" est chargée de faire le véritable traitement de la requête en effectuant les calculs adéquats et rendre le résultat. A cette effet, elle a pour entrée le PID du client, la taille des données à traiter et l'opération à appliquer sur ces données. Le PID du client sert comme clé pour pouvoir accéder à la mémoire partagé avec le client et qui comporte deux champ : le tableau de données et un entier servant pour la synchronisation entre le client et le serveur et pour indiquer si le serveur (worker) a fini les calculs et a déposé le résultats dans cette mémoire partagée. Les calculs réalisées par cette fonction suivent l'algorithme de Hills Steel Scan.

Cette fonction effectue les traitement suivants:

- Étape 1 : fait un appel système (***shmget()***) en lui fournissant le PID du client comme clé et cela pour obtenir l'id du segment de mémoire partagée avec le client qui a envoyé une requête.
- Étape 2 : attache (***shmat()***) à son espace d'adressage le segment de mémoire partagée obtenu pour avoir un pointeur vers sa zone.

- Etape 3 : Fait les calculs sur les données du client selon l'algorithme de Hills stelle scan comme suit:
    - création d'un tableau qui va contenir tous les numéros de threads qui seront créés.
    - initialisation des tableaux *data* et *data_new* aux données initiales (se trouvant en mémoire partagée)
    - calcul du nombre d'étapes (*nbEtapes*) à faire (selon l'algorithme de Hills steel scan)
    - boucle de calcul dont le nombre d'itération est égale aux nombre d'étape calculé auparavant,
    - Dans chaque itération de cette boucle : lancement de plusieurs threads en parallèle. Pour chaque thread, on indique un indice et un numéro d'étape. A base de ces 2 informations le thread va appliquer une opération sur deux éléments du tableau de données : le premier est identifié par l'indice fourni et le second élément est identifié par l'indice calculé à base de l'indice fourni et du numéro de l'étape courante (indice - $2^{\text{étape courante}}$).

Remarque importantes:

- Afin d'éviter des problèmes de passage de paramètres aux threads sachant que ces derniers opèrent en parallèle à chaque étape de l'algorithme de Hills Steel Scan, nous avons pris la précaution de créer (malloc) à chaque itération une nouvelle instance de la structure fournie en paramètre, que nous libérons (free) après usage.
- Toujours, afin d'éviter des interférences entre les threads, nous avons pris la précaution de récupérer les identifiants des threads dans un tableau dédié à cela. Ainsi, nous pourrons mieux contrôler leur synchronisation. C'est comme cela, que nous avons pu à chaque étape de l'algorithme, définir un point de synchronisation (pthread_join) à chaque fin d'étape. Cela veut dire que les étapes sont exécutées séquentiellement l'une après l'autre.

Le programme de contrôle du serveur

Le but de ce programme est de contrôler le lancement et l'arrêt du serveur.
Lors du lancement, il s'assure que le serveur n'est pas déjà lancé et lors de l'arrêt, qu'il n'est pas déjà arrêté.
Lorsqu'on quitte ce programme, il arrête le processus serveur s'il est en exécution.